Exam 1

- Exam 1 on Friday, October 6
- Closed book, closed computers/phones
- You are allowed one “cheat page”, a standard 8.5x11-sized sheet filled on one side
  - Type or write by hand (which I recommend)

Topics

- Reasoning about code
  - Forward and backward reasoning, logical conditions, Hoare triples, weakest precondition, rules for assignment, sequence, if-then-else, loops, loop invariants, decrementing functions

Forward Reasoning

- Forward reasoning simulates the execution of code. Introduces facts as it goes along
  - E.g., \( \{ x = 1 \} \)
    \[
    \begin{align*}
    & y = 2 \times x \\
    & \{ x = 1 \land y = 2 \} \\
    & z = x + y \\
    & \{ x = 1 \land y = 2 \land z = 3 \}
    \end{align*}
    \]
  - Collects all facts, often those facts are irrelevant to the goal

Backward Reasoning

- Backward reasoning “goes backwards”. Starting from a given postcondition, finds the weakest precondition that ensures the postcondition
  - E.g., \( \{ y < 1 \} \) // Simplify \( 2y < y+1 \) into \( y < 1 \)
    \[
    \begin{align*}
    & z = y + 1 \quad \text{// Substitute } y+1 \text{ for } z \text{ in } 2y < z \\
    & \{ 2y < z \} \\
    & x = 2y \quad \text{// Substitute } rhs \text{ for } x \text{ in } x < z \\
    & \{ x < z \}
    \end{align*}
    \]
  - More focused and more useful

Condition Strength

- “P is stronger than Q” means “P implies Q”
- “P is stronger than Q” means “P guarantees more than Q”
  - E.g., \( x > 0 \) is stronger than \( x > -1 \)
  - Fewer values satisfy P than Q
    - E.g., fewer values satisfy \( x > 0 \) than \( x > -1 \)
- Stronger means more specific
- Weaker means more general
Condition Strength

• Which one is stronger?
  \( x > -10 \) or \( x > 0 \)
  \( x > 0 && y = 0 \) or \( x > 0 || y = 0 \)
  \( 0 \leq x \leq 10 \) or \( 5 \leq x \leq 11 \) (Neither!)
  \( y = 2 \text{ (mod 4)} \) or \( y \) is even
  \( x = 10 \) or \( x \) is even

Hoare Triples

• A Hoare Triple: \{ \( P \) \} code \{ \( Q \) \}
  - \( P \) and \( Q \) are logical statements about program values, and code is program code (in our case, Java code)
  - \{ \( P \) \} code \{ \( Q \) \} means “if \( P \) is true and we execute code, then \( Q \) is true afterwards”
  - \{ \( P \) \} code \{ \( Q \) \} is a logical formula, just like “index"

Examples of Hoare Triples

\{ \( x > 0 \) \} \( x++ \) \{ \( x > 1 \) \} is true
\{ \( x > 0 \) \} \( x++ \) \{ \( x > -1 \) \} is true
\{ \( x \geq 0 \) \} \( x++ \) \{ \( x > 1 \) \} is false. Why?

\{ \( \neq 0 \) \} \( x++ \) \{ \( x > 0 \) \} is ??
\{ \( \neq 0 \) \} \( x++ \) \{ \( x > 0 \) \} is ??
\{ \( \neq 0 \) \} \( x++ \) \{ \( x > 0 \) \} is ??
\{ \( \neq 0 \) \} \( x++ \) \{ \( x > 0 \) \} is ??
\{ \( x = a \) \} if \( x < 0 \) \( x = -x \) \{ \( x = | a | \) \} is ??
\{ \( x = y \) \} \( x = x + 3 \) \{ \( x = y \) \} is ??

Rules for Backward Reasoning:

Assignment

// precondition: ??
\( x = \) expression
// postcondition: Q

Rule: the weakest precondition = Q, with all occurrences of \( x \) in Q replaced by expression

More formally:
wp("x=expression","Q") = Q with all occurrences of x replaced by expression

Sequence

// precondition: ??
\( S1 \); // statement
\( S2 \); // another statement
// postcondition: Q

Work backwards:
precondition is wp("S1;S2","Q") = wp("S1;","wp("S2","Q")")

Example:
// precondition: ??
\( x = 0 \);
\( y = x+1 \);
// postcondition: \( y > 0 \)

Rules for If-then-else

Forward reasoning
Backward reasoning
\{ \( P \) \}
\{ \( \bigwedge b \wp("S1","Q") \lor \neg b \wp("S2","Q") \) \}
if \( b \)
if \( b \)
\{ \( P \land b \) \}
\{ \wp("S1","Q") \}
\( S1 \)
\( S1 \)
\( Q1 \)
\( Q \)
else
else
\{ \( P \land \neg b \) \}
\{ \wp("S2","Q") \}
\( S2 \)
\( S2 \)
\( Q2 \)
\( Q \)
\( Q1 \lor Q2 \)
\( Q \)
Reasoning About Loops by Induction

1. Partial correctness
   - Guess loop invariant
   - Prove loop invariant using computation induction
   - Loop exit condition (i.e., !b) and loop invariant imply the desired postcondition
2. Termination
   - (Intuitively) Guess "decrementing function" D:
     1) D is bounded, 2) at each iteration D decreases, and 3) D = 0 AND loop invariant, imply loop exit condition (i.e., D = 0 AND loop invariant imply !b)

Example: Partial correctness

Precondition: x >= 0;
i = x;
z = 0;
while (i != 0) {
z = z+1;
i = i-1;
}
Postcondition: x = z;

Example: Termination

Precondition: x >= 0;
i = x;
z = 0;
while (i != 0) {
z = z+1;
i = i-1;
}
Postcondition: x = z;

Reasoning About Loops

1) \(i=x\) and \(z=0\) give us that \(i+z=x\) holds at 0th iteration of loop // Base case
2) Assuming that \(i+z=x\) holds after \(k\)th iteration, we show it holds after \((k+1)\)th iteration // Induction
   \[z_{new} = z + 1\] and \(i_{new} = i - 1\) thus
   \[z_{new} + i_{new} = z + 1 + i - 1 = z + i = x\]
3) If loop terminated, we know \(i = 0\).
   Since \(z+i=x\) holds, we have \(x = z\)

Topics

- Specifications
  - Benefits of specifications, PoS specification convention, specification style, specification strength (stronger vs. weaker specifications), comparing specifications via logical formulas, converting PoS specifications into logical formulas

Fall 17 CSCI 2600, A Milanova
Specifications

A specification consists of a **precondition** and a **postcondition**
- **Precondition**: conditions that hold before method executes
- **Postcondition**: conditions that hold after method finished execution (if precondition held!)

Example Specification

**Precondition:** len ≥ 1 && arr.length = len
**Postcondition:** returns arr[0]+...+arr[arr.length-1]

```java
double sum(int[] arr, int len) {
    double sum = arr[0];
    int i = 1;
    while (i < len) {
        sum = sum + arr[i];
        i = i+1;
    }
    return sum;
}
```

Benefits of Specifications

- Precisely documents method behavior
- Imagine if you had to read the code of the Java libraries to figure what they do!
- An abstraction --- abstracts away unnecessary detail
- Promotes modularity
- Enables reasoning about correctness
- Through testing and/or verification

PoS Specifications

- Specification convention due to Michael Ernst
  - **The precondition**
    - **requires**: clause spells out constraints on client
  - **The postcondition**
    - **modifies**: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
    - **throws**: lists possible exceptions
    - **effects**: describes final state of modified objects
    - **returns**: describes return value

Example

```java
static List<Integer> listAdd(List<Integer> lst1, List<Integer> lst2) {
    List<Integer> res = new ArrayList<Integer>();
    for (int i = 0; i < lst1.size(); i++)
        res.add(lst1.get(i) + lst2.get(i));
    return res;
}
```
**Another Example**

```java
static void listAdd2(List<Integer> lst1,
                    List<Integer> lst2) {
    requires: lst1, lst2 are non-null.
    lst1 and lst2 are same size.
    modifies: lst1
    effects: i-th element of lst1 is replaced with the sum of i-th
            elements of lst1 and lst2
    returns: none
}
```

**Specification Style**

- A method is called for its side effects (effects clause) or its return value (returns clause)
- It is bad style to have both effects and return
- There are exceptions
  - E.g., HashMap.put returns the previous value
- Main point of spec is to be helpful
  - Being overly formal does not help
  - Being too informal does not help either
- If spec turns too complex: redesign. Better to simplify code than document complexity!

**What’s Wrong?**

```java
static void uniquefy(List<Integer> lst) {
    requires: ?
    modifies: ?
    effects: ?
    returns: ?
}
```

**Specification Strength**

- “A is stronger than B” means
  - For every implementation I
    - “I satisfies A” implies “I satisfies B”
    - The opposite is not necessarily true
  - For every client C
    - “C works with B” implies “C works with A”
    - The opposite is not necessarily true
- Principle of substitutability:
  - A stronger spec can always be substituted for a weaker one

**Why Care About Specification Strength?**

- Because of substitutability!
- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - i.e., an implementation that conforms to a stronger specification can be used in a client that expects a weaker specification

**Substitutability**

- Substitutability guarantees correct software updates, correct class hierarchies
- Client code: `X x; ... x.foo(index);`
- Client is “polymorphic”: written against `X`, but it is expected to work with any subclass of `X`
- A subclass of `X`, say `Y`, may have its own implementation of `foo`, `Y.foo(int)`. Client
  must work correctly with `Y.foo(int)`!
- If spec of `Y.foo(int)` is stronger than that of `X.foo(int)` then we can safely substitute `Y.foo(int)` for `X.foo(int)`!
Strengthening and Weakening Specification

- **Strengthen a specification**
  - Require less of client: fewer conditions in requires clause AND/OR
  - Promise more to client: effects, modifies, returns

- **Weaken a specification**
  - Require more of client: add conditions to requires AND/OR
  - Promise less to client: effects, modifies, returns clauses are weaker, thus easier to satisfy in code

Comparing by Logical Formulas

- If satisfies specification A) is a logical formula: \( P_A \Rightarrow Q_A \)
  \( P_A \) is the precondition of A, \( Q_A \) is the postcondition of A

- Spec A is **stronger** than spec B if and only if for each implementation \( I \), \( (I \text{ satisfies } A) \Rightarrow (I \text{ satisfies } B) \) which is equivalent to \( A \Rightarrow B \)

- A is stronger than B iff \( (P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) \)

Recall from FoCS and/or Intro to Logic: \( p \Rightarrow q \equiv \neg p \lor q \)

Exercise: Order by Strength

- Spec A: **requires**: a non-negative int argument
  **returns**: an int in [1..10]

- Spec B: **requires**: int argument
  **returns**: an int in [2..5]

- Spec C: **requires**: true
  **returns**: an int in [2..5]

- Spec D: **requires**: an int in [1..10]
  **returns**: an int in [1..20]

Compare Spec to Formula, step 1: absorb throws and returns into effects

```java
set from java.util.ArrayList<T>
T set(int index, T element)
requires: true
modifies: this[index]
effects: this.set(index) = element
throws: IndexOutOfBoundsException if index < 0 || index ≥ size
returns: this.size(index)
```

Absorb effects, returns and throws into new **effects**: if index < 0 || index ≥ size then throws IndexOutOfBoundsException else this.post[index] = element and returns this.pre[index]

Converting PoS Specs into Logical Formulas

- PoS specification
  - **requires**: R
  - **modifies**: M
  - **effects**: E // absorbs throws, returns and effects

Spec is equivalent to the following logical formula:
\( R \Rightarrow (E \&\& (\text{nothing but } M \text{ is modified})) \)

Step 1: absorb throws and returns into effects E
Step 2: write \( R \Rightarrow (E \&\& (\text{nothing but } M \text{ is modified})) \)
Convert Spec to Formula, step 2: Convert into Formula

```java
T set(int index, T element)
```

requires: true
modifies: this[index]
effects: if index < 0 || index ≥ size then
        throws IndexOutOfBoundsException
        else
        this[i] = element and returns this[i]

Denote effects expression by E. Resulting formula is:

```
true => (E ∧ (foreach i ≠ index, this[i] = this[i]))
```

ADTs

- Abstract Data Type (ADT): higher-level data abstraction
  - The ADT is operations + object
  - A specification mechanism
  - A way of thinking about programs and design

An ADT Is a Set of Operations

- Operations operate on data representation
- ADT abstracts from organization to meaning of data
- ADT abstracts from structure to use
- Data representation does not matter!

```java
class Point {
    float x, y;
}
```

- Instead, think of a type as a set of operations: create, x(), y(), r(), theta().
- Force clients to call operations to access data

Specifying an ADT

<table>
<thead>
<tr>
<th>immutable</th>
<th>mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>class TypeName</td>
<td>class TypeName</td>
</tr>
<tr>
<td>1. overview</td>
<td>1. overview</td>
</tr>
<tr>
<td>2. abstract fields</td>
<td>2. abstract fields</td>
</tr>
<tr>
<td>3. creators</td>
<td>3. creators</td>
</tr>
<tr>
<td>4. observers</td>
<td>4. observers</td>
</tr>
<tr>
<td>5. producers</td>
<td>5. producers (rare!)</td>
</tr>
<tr>
<td>6. mutators</td>
<td>6. mutators</td>
</tr>
</tbody>
</table>

Connecting Implementation to Specification

- Representation invariant: Object → boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values

- Abstraction function: Object → abstract value
  - What the data structure really means
    - E.g., array [2, 3, -1] represents \(-x^2 + 3x + 2\)
  - How the data structure is to be interpreted
Representation Exposure

- Client can get control over rep and break the rep invariant! Consider
  \[ \text{IntSet } s = \text{new IntSet}(); \]
  \[ \text{s.add}(1); \]
  \[ \text{List}\langle\text{Integer}\rangle \text{ li} = \text{s.getElements}(); \]
  \[ \text{li.add}(1); // Breaks IntSet's rep invariant! \]
- Representation exposure is external access to the rep. **AVOID!!**
- If you allow representation exposure, document why and how and feel bad about it.

```
public List\<Integer\> getElements() {
    return new ArrayList\<Integer\>(data);
}
```

Representation Exposure

- Make a copy on the way out:
  \[ \text{public List}\langle\text{Integer}\rangle \text{ getElements}() \{ \]
  \[ \text{    return new ArrayList}\langle\text{Integer}\rangle(\text{data}); \}
  \[ \text{\}
  \]
- Mutating a copy does not affect IntSet’s rep
  \[ \text{IntSet } s = \text{new IntSet}(); \]
  \[ \text{s.add}(1); \]
  \[ \text{List}\langle\text{Integer}\rangle \text{ li} = \text{s.getElements}(); \]
  \[ \text{li.add}(1); // mutates new copy, not IntSet's rep \]

```
public IntSet(\text{ArrayList}\langle\text{Integer}\rangle \text{ elts}) { 
    \text{data} = \text{new ArrayList}\langle\text{Integer}\rangle(\text{elts});
    ... 
}
```

Abstraction Function

- Abstraction function allows us to reason about correctness of the implementation

```
IntSet Example

Creating concrete object: Establish rep invariant
Establish abstraction function
After every operations: Maintains rep invariant
Maintains abstraction function
```